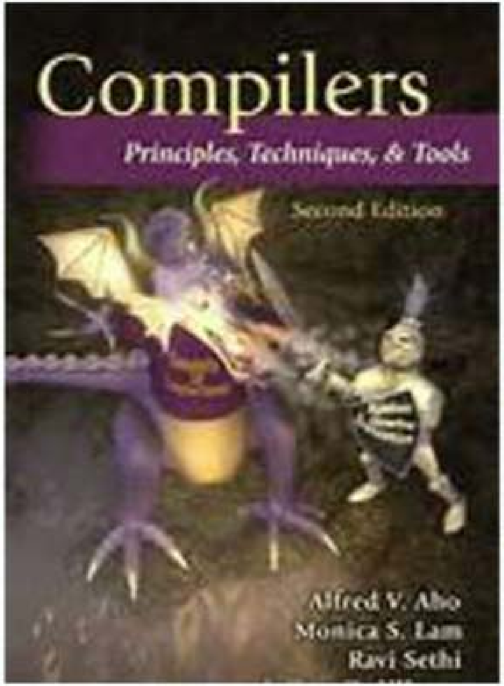
RyuJIT

.NET Dynamic Code Execution:

New Platforms, Better Code, and Adaptive Optimization

Why "RyuJIT"?

- Ryujin is a Japanese Sea Dragon
- The name RyuJIT is a geeky nod to the "Dragon Book"



Agenda

- A brief history of the .NET JIT
- RyuJIT Vision
- Architectural Principles
- Front end
  - SSA and Value Numbering
- Backend
  - Lowering, Linear Scan Register Allocation, and Code Selection
- Status
- Future

1996: x86 JIT

- x86 is the only architecture that matters
- CPU’s are always getting faster
- Result:
  - JIT32: decent code quality, small package

2001: .NET on Itanium

- Servers need Code Quality
- Use the C++ optimizer
  - IA64 is hard to JIT
  - Servers don’t care about startup time
  - Servers have lots of RAM
- Result:
  - IA64 JIT works
  - O(N^X) algorithms, in both time & space, pervade

2003: .NET on AMD64

- 64 bits are for servers
- Just port the IA64 JIT
- Results:
  - Same general characteristics as IA64
  - But people now use this thing for workstations!
  - … and desktops
  - … and Surface Pro's

2010: .NET on ARM

- Slow CPU, less RAM: just port JIT32
- Gack: ARM32 is NOTHING like x86!
- Results:
  - About 12 dev years of work
  - Horrible code quality
  - Relatively bad compile speeds, too…

2012: .NET everywhere!

- X86 JIT is fragile (JIT32)
- X64 JIT is messy and slow (JIT64)
- ARM32 code quality is terrible and difficult to address (JIT32)
- ARM64 is coming: which code base?
- Result:
  - RyuJIT (aka JITBlue) green light

RyuJIT: The Vision

Offer developers a high quality JIT compiler that delivers good throughput, code quality and feature & fix turn-around time, with consistent, predictable performance across all architectures.

Vision-Focused Architecture

- Offer developers a high quality JIT compiler …
- That delivers good throughput …
- Code quality …
- And feature & fix turn-around time …
- With consistent, predictable performance across all architectures.

- Modern compiler architecture
- Linear or near-linear compile-time algorithms
- SSA-based optimizations
- VN designed to leverage type safety
- Goal of “dialable” CQ vs. throughput
- Regularize the IR and strengthen the invariants
- Single code base for new features, e.g. SIMD
- Eliminate implicit x86-focused assumptions
- Isolate architecture-specifics in Lowering and code selection

RyuJIT Plan: Evolve JIT32

- Focus on areas where real value can be derived
- Discourage rewriting for rewriting's sake
- Encourage review of significant problems with JIT32 IR
- First deliverable: x64 JIT
  - Biggest bang-for-the-buck for customers
  - Servers care about startup time, memory usage, AND code quality!

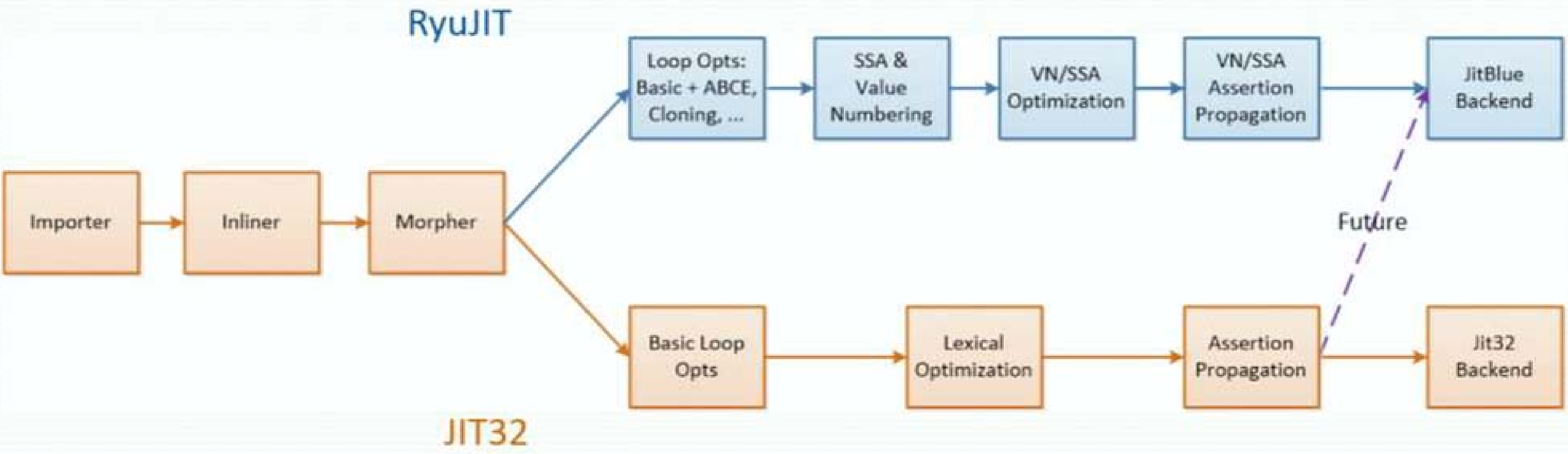
Evolution Strategy

- Reuse basic JIT32 IR and structure:
  - Modest architectural trade-offs relative to the ideal from-scratch
  - Significant risk reduction in both functional and bug-for-bug compatibility
  - Deliver measurable benefits in phases without "going dark"
  - Phased-in "rationalization" of the trees
  - More Object orientation
    - STL-style collections and iterators
    - More abstraction and encapsulation
- Optimizations mostly done on high-level (roughly IL-equivalent) trees
  - Favor semantic analysis over pattern matching
- Register allocation and code generation done on IR with all target requirements explicit
  - Lowering phase transforms high-level to low-level IR in-place (many trees remain unchanged)
    - Tree structure preserved, but nodes are linearized for backend traversal

Evolution Tactics

- Keep
  - Fundamental Tree/IR structure
  - Importation and Inlining
- Change
  - Rationalize the tree representation (phased-in, initially BE only)
    - Restructure problematic nodes (ASG, ADDR, QMARK, COMMA, CALL)
    - Tree orientation in FE, linear ordering dominant in BE
  - SSA and VN based optimizations
  - Linear scan register allocation
  - Linear code generation

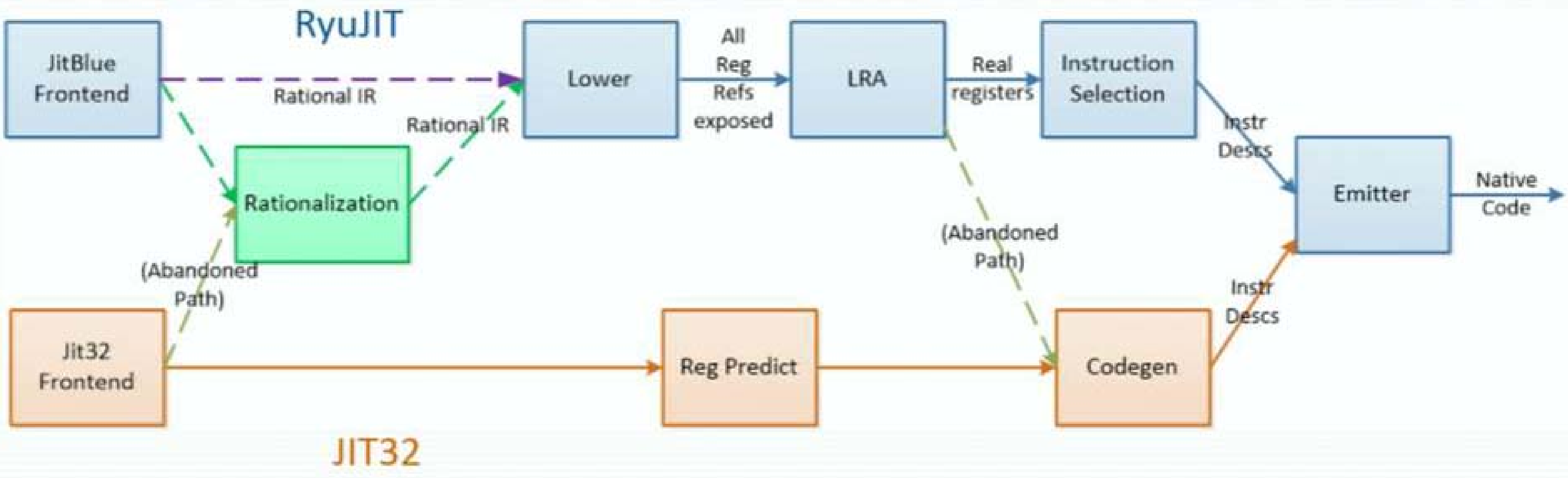
Front End


RyuJIT
Loop Opts: Basic + ABCE, Cloning, ...
SSA & Value Numbering
VN/SSA Optimization
VN/SSA Assertion Propagation
JitBlue Backend
Importer
Inliner
Morpher
Future
Basic Loop Opts
Lexical Optimization
Assertion Propagation
Jit32 Backend
JIT32

SSA

- SSA renaming for all local variables and temps
- SSA-based Copy Propagation
- SSA forms the basis for Value Numbering

RyuJIT Backend



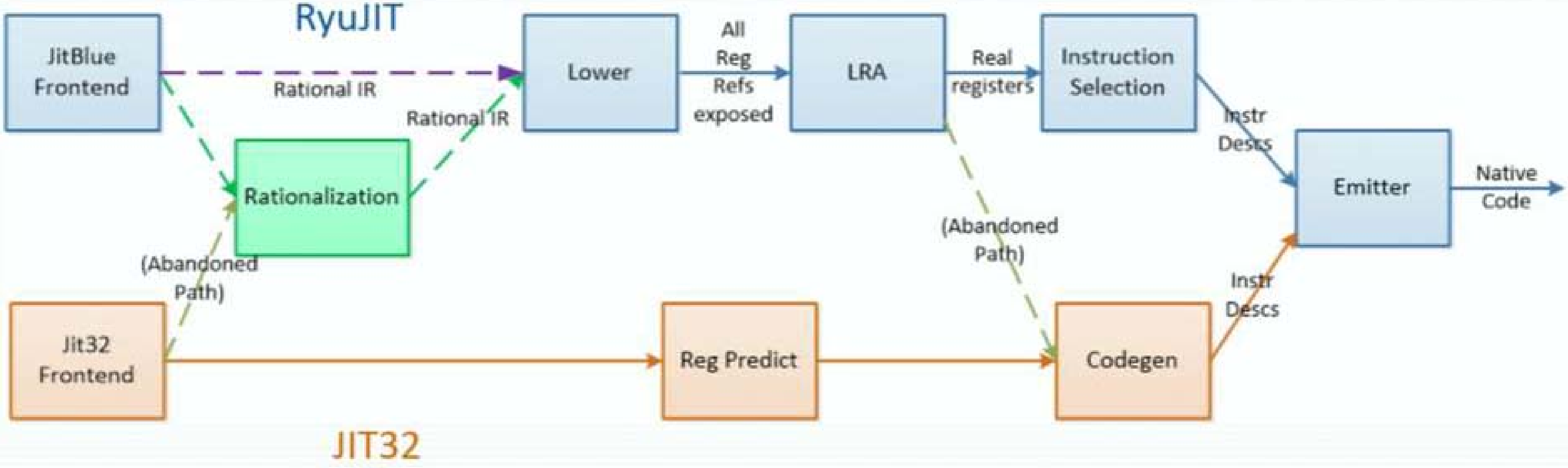
SSA

- SSA renaming for all local variables and temps
- SSA-based Copy Propagation
- SSA forms the basis for Value Numbering

Value Numbering

- Built on top of SSA, but not just variables
  - Expressions
  - Heap value numbering using field maps, utilizing type safety
- Assertion Propagation
- CSE
  - Value rather than lexical equivalence
- Loop invariant code motion
- Array bounds check elimination

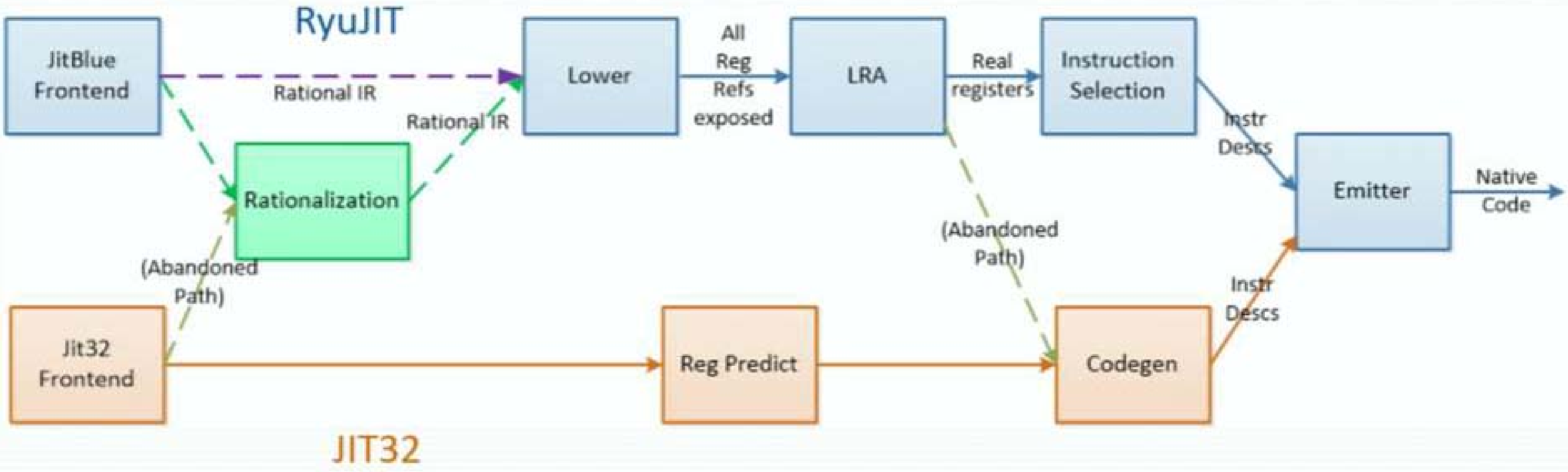
RyuJIT Backend



Value Numbering

- Built on top of SSA, but not just variables
  - Expressions
  - Heap value numbering using field maps, utilizing type safety
- Assertion Propagation
- CSE
  - Value rather than lexical equivalence
- Loop invariant code motion
- Array bounds check elimination

RyuJIT Backend



Lowering

- Transforms IR prior to register allocation and code generation
  - Linearized IR nodes (execution order)
  - Fully exposed register requirements
- "Logically atomic"
  - Not necessarily 1:1 mapping between nodes and target instructions
    - Register lifetimes must be exposed and accurately modeled
  - Some registers may be "internal" (used only during the execution of the node)

New Register Allocator

- JIT32 register allocator uses coloring for locals, while "predicting" register requirements for expression evaluation
- Complex, error-prone with too much implicit hand-shaking between it and codegen

Linear Scan Register Allocation

- Much better scaling
- Algorithmic rather than ad-hoc
  - Target-dependent phase determines register requirements per node
  - Allocation is performed over IR-independent register references
  - Unified treatment of locals and expression temporaries
- "Standard" linear scan, with splitting
- Challenges
  - Block ordering and resolution (work in progress)
  - Spill locations
- Future
  - Tuning, tuning, tuning
  - Utilization of SSA for better lifetime splitting and enhanced copy propagation

Engineering Methodologies

- Fall back JIT during x64 development
  - “NYI” mechanism reverted to JIT64 when unimplemented features were encountered
- SuperPMI
  - Record and replay of JIT/EE interface to enable much faster checking for JIT-time failures in test suites
- Stress modes for the register allocator
  - Essential for testing “corner case” handling that otherwise occurs rarely
    - Constrain number and type (caller/callee) of available regs
    - Modify selection heuristics for both free and spilling
    - Spill-always
    - Extend lifetimes
    - Modify block traversal order and boundary locations

First Instantiation: RyuJIT<x64>

- Feature parity; very few outstanding bugs
- Currently in its second CTP release!
  - http://blogs.msdn.com/b/dotnet/archive/2014/02/27/ryujit-ctp2-getting-ready-for-prime-time.aspx

RyuJIT Compilation Speed

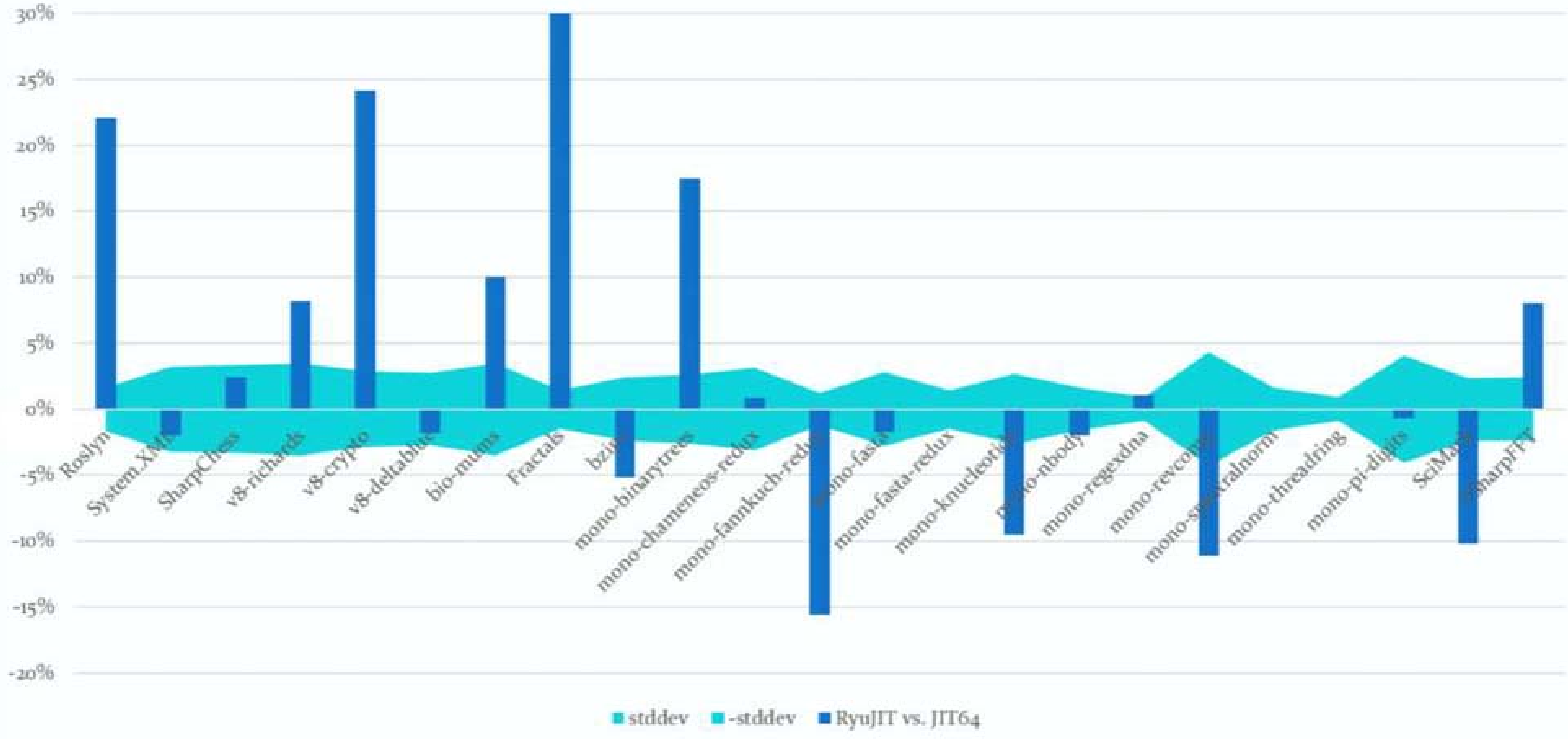
- RyuJIT trounces JIT64 handily
  - Memory consumption is generally less than half
  - Throughput is, on average, 2X
  - Corner Cases look much, much better (40X better)
- X64 RyuJIT still slower than x86 JIT32
  - Haven’t spent too much time here
  - May do so in the future.

RyuJIT Code Quality

- For C# “Real Word Code”, we’re generally better
  - Plenty of outliers still exist
- Certain classes of loops we generally lose on (for the moment)
- Smaller benchmarks are hit & miss.

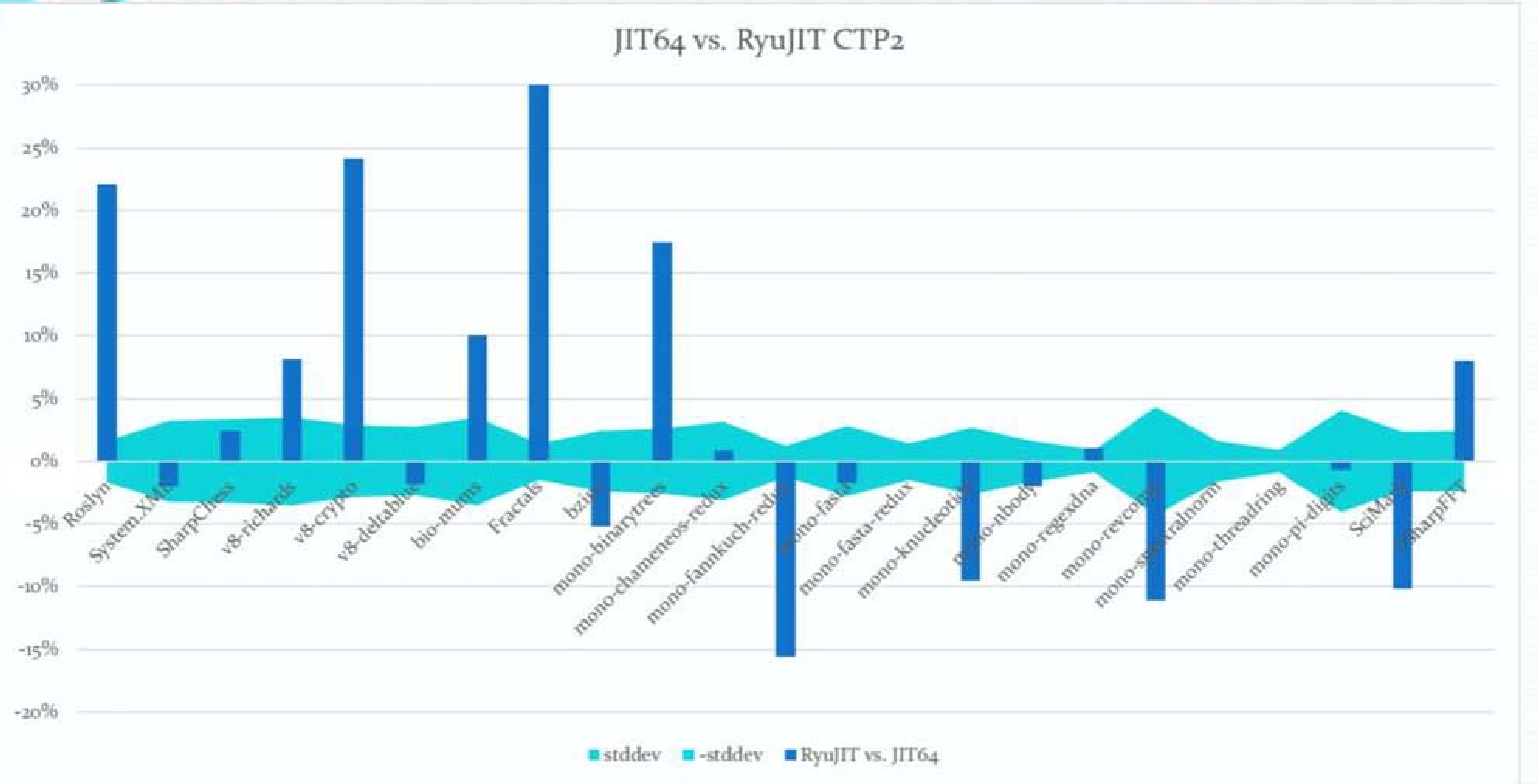
(Eye Candy on the next slide)

JIT64 vs. RyuJIT CTP2



Current Development

- Loop Cloning
- SIMD
    - CPU agnostic, length-variable generic vector types:
        - Vector<T>
    - Plus common fixed-length types, e.g. Vector2f, Vector3f, etc.
    - Coming out party in April
    - Already notified MVP's it's coming
- RyuJIT<ARM64>
    - Hardware will be here "real soon"
- ProjectN JIT



JIT64 vs. RyuJIT CTP2
30%
25%
20%
15%
10%
5%
0%
-5%
-10%
-15%
-20%
Roslyn
System.XML
SharpChess
v8-richards
v8-crypto
v8-deltablue
bio-mums
Fractals
bzip2
mono-binarytrees
mono-chameneos-redux
mono-fannkuch-redux
mono-fasta
mono-fasta-redux
mono-knucleotide
mono-nbody
mono-regexdna
mono-revcomp
mono-spectralnorm
mono-threadring
mono-pi-digits
SciMark
SharpFFT
stddev
-stddev
RyuJIT vs. JIT64

RyuJIT Code Quality

- For C# “Real Word Code”, we’re generally better
  - Plenty of outliers still exist
- Certain classes of loops we generally lose on (for the moment)
- Smaller benchmarks are hit & miss.

(Eye Candy on the next slide)

Current Development

- Loop Cloning
- SIMD
  - CPU agnostic, length-variable generic vector types:
    - Vector<T>
  - Plus common fixed-length types, e.g. Vector2f, Vector3f, etc.
  - Coming out party in April
  - Already notified MVP's it's coming
- RyuJIT<ARM64>
  - Hardware will be here "real soon"
- ProjectN JIT

Longer term future work

- Dynamic optimization
  - Server workloads demand it
  - Competition is already do a lot
  - Incubation, with big plans
- RyuJIT x86
  - Servers like 32 bits, too!
- SIMD/Data Parallel opts: looking at feasibility & utility
  - PLINQ => SIMD/GPGPU
  - Long length vectors
- IL => IL optimizations?

Dynamic Opts: Characteristics

- Utilize dynamic information (types, behaviors)
- Leverage the ability to defer or redo compilation
- May be speculative (guarded)
  - Uses information (or predictions) that might be invalidated dynamically
  - Defer code generation of anything post-dominated by a throw, or otherwise predicted to be rare
- May be costly
  - Only worth applying to “hot” methods

Dynamic Information

- Information about the type system
  - Class hierarchy
  - Properties of methods (read/write heap, synch, run once …)
  - Properties of method call sites (parameter info)
  - Properties of fields (initonly)
- Information about observed behavior (profile)
- “Static” information that’s available only AFTER Jitting
  - Post-morphing size for future inlining consideration (e.g. after outlining exception paths, or intrinsic calls that turn into small code)
  - Pure function, heap accesses (see properties of methods, above)

Loop Cloning

- Multi-versioning
  - Pre-checked
  - Vectorization
- Deferred generation of alternate loop
  - If known to be rare

Devirtualization

- Change interface dispatch to vtable-like dispatch
- Change virtual calls into (possibly checked) direct calls
  - Including interface dispatch
- Start with interface dispatch, and use stubs until you know a vtable slot is worth it
- Main benefit: exposing additional inlining opportunities

Generic Specialization

- Eliminating dictionary lookups
- Specializing for reference types
  - Developer-specified
  - As encountered (profile)
  - Where there’s a switch on the type

Method Cloning

- Useful when inlining may be too costly
- Specialized method for known parameter type(s) or characteristics, e.g.
  - Parameter reference non-equivalence
  - Shape equivalence, indices in-bounds
- Enables, and enabled by, other optimizations such as devirtualization